Procesar un archivo de 500GB con un portátil utilizando Ray Datasets

Índice

Introducción.

Imagina que tengo un archivo de 50 GB guardados en un usb de mi equipo, de nombre "archivo50G.json". Me gustaría que utilizando **Ray Datasets**, para procesar el archivo para crear unos cálculos con alguno de sus datos (columnas)

Vamos a hacerlo paso a paso pensando en tu W11, un USB y un archivo gordo tipo archivo50G.json.

Vamos a asumir:

- Estamos en **Windows 11**.
- Tienes **Ray instalado** en tu entorno (pip install "ray[data]" o ray[default]).

```
pip install "ray[data]"
# o bien
pip install "ray[default]"
```

- El JSON es **newline-delimited** (JSON Lines): *un objeto JSON por línea*. Ejemplo:
- `{"x": 1.2, "y": 3.4}`
- `{"x": -0.5, "y": 0.8}`
- ...

1. Localizar el archivo en el USB

En Windows, el USB suele tener una letra de unidad, por ejemplo E: o F:.

Imaginemos:

E:\datos\archivo50G.json

En Python, mejor usar una cadena raw:

ruta_archivo = r"E:\datos\archivo50G.json"

2. Script base con Ray Datasets

Te pongo primero el código completo, luego lo troceamos y explicamos:

```
import os
import time
import ray
import ray.data as rd
import pandas as pd

# ---------------------------------------------------------------------
# 0. Punto de entrada (MUY IMPORTANTE en Windows)
# ---------------------------------------------------------------------
if __name__ == "__main__":

    # 1. Inicializar Ray usando todos los núcleos lógicos
    ray.shutdown()
```

```
ray.init(num_cpus=os.cpu_count())
print("CPUs lógicas:", os.cpu_count())

# 2. Ruta al JSON gigante en el USB
ruta_archivo = r"E:\datos\archivo50G.json"  # ajusta esto a tu ruta real

# 3. Leer el JSON como Ray Dataset (NO se carga todo en memoria de golpe)
inicio = time.time()
ds = rd.read_json(ruta_archivo)   # Ray se encarga de partir el fichero en bloques
fin = time.time()
print("Dataset leído.")
print(ds)
print(f"Tiempo de read_json: {fin - inicio:.2f} s")

# 4. Ver unas pocas filas para inspeccionar (no carga todo)
print("\nPrimeras filas:")
ds.show(5)

# 5. Definir una función de transformación por batch (pandas DataFrame)
def enrich_batch(batch: pd.DataFrame) -> pd.DataFrame:
    # Importante: trabajar sobre una copia si vamos a añadir columnas
    batch = batch.copy()

    # Ejemplos de transformación:
    # - Nueva columna z = x^2 + y^2 (como antes)
    if "x" in batch.columns and "y" in batch.columns:
        batch["z"] = batch["x"]**2 + batch["y"]**2

        # por ejemplo, media simple de x e y
        batch["xy_mean"] = (batch["x"] + batch["y"]) / 2

    # Puedes añadir más lógica, filtros, etc.
    return batch

# 6. Aplicar la transformación en paralelo con Ray sobre todos los datos
inicio = time.time()
ds_enriched = ds.map_batches(
    enrich_batch,
    batch_format="pandas",  # cada batch será un DataFrame de pandas
)
fin = time.time()
print(f"\nTransformación paralela completada en: {fin - inicio:.2f} s")
print(ds_enriched)

# 7. Ejemplo de agregados globales: media de alguna columna
if "z" in ds_enriched.schema().names:
    mean_z = ds_enriched.mean("z")
    print(f"\nMedia global de z: {mean_z}")

if "xy_mean" in ds_enriched.schema().names:
    mean_xy_mean = ds_enriched.mean("xy_mean")
    print(f"Media global de xy_mean: {mean_xy_mean}")

# 8. Guardar el resultado enriquecido en disco de forma particionada (parquet)
salida_dir = r"E:\datos\salida_enriquecida" # o en tu disco interno, mejor que en el USB
os.makedirs(salida_dir, exist_ok=True)

inicio = time.time()
ds_enriched.write_parquet(salida_dir)
fin = time.time()
print(f"\nDatos enriquecidos guardados en: {salida_dir}")
print(f"Tiempo de escritura parquet: {fin - inicio:.2f} s")

ray.shutdown()
```

3. Qué está pasando en cada paso

3.1. Punto de entrada para Windows

```
if __name__ == "__main__":
    ...
```

En Windows, cuando usas procesos (Ray los usa por debajo), **esto es obligatorio** para que no se creen procesos recursivos.

Dentro de ese bloque va todo el código "real".

3.2. Inicializar Ray

```
ray.shutdown()
ray.init(num_cpus=os.cpu_count())
print("CPUs lógicas:", os.cpu_count())
```

- ray.shutdown() limpia cualquier instancia anterior.
- ray.init(num_cpus=...) arranca Ray usando todos tus núcleos lógicos.
- Imprimimos cuántos hay (en tu caso eran 8).

3.3. Ruta al JSON en el USB

```
ruta_archivo = r"E:\datos\archivo50G.json"
```

- Ajustas la letra de unidad y la carpeta a lo que tengas realmente.
- Usa `r"..."` para evitar problemas con barras invertidas en Windows.

3.4. Leer el JSON como Ray Dataset

```
inicio = time.time()
ds = rd.read_json(ruta_archivo)
fin = time.time()
print("Dataset leído.")
print(ds)
print(f"Tiempo de read_json: {fin - inicio:.2f} s")
```

- rd.read_json(...):
    - No crea un único DataFrame gigante en memoria.
    - Crea un **Ray Dataset**, que está dividido en bloques.
    - Ray puede leer el fichero por trozos y repartirlos entre workers.

Ojo: lo ideal es que el JSON sea de tipo **JSON Lines** (un objeto por línea). Si tu fichero es un único mega-array de JSON ([{...}, {...}, ...]), hay que tratarlo diferente.

Si intentas leer un JSON gigantesco que es un único array [...], Ray puede necesitar leerlo entero para parsearlo, y ahí sí te puedes quedar sin RAM. Siempre que se pueda, conviene convertirlo antes a JSON Lines o directamente a parquet.

3.5. Ver algunas filas

```
print("\nPrimeras filas:")
ds.show(5)
```

- `ds.show(5)`:
    - Solo muestra 5 filas.
    - No trae los 50 GB a memoria.
- Sirve para comprobar que las columnas (`x`, `y`, etc.) están como esperas.

3.6. Función de transformación por batch

```
def enrich_batch(batch: pd.DataFrame) -> pd.DataFrame:
    batch = batch.copy()

    if "x" in batch.columns and "y" in batch.columns:
        batch["z"] = batch["x"]**2 + batch["y"]**2
        batch["xy_mean"] = (batch["x"] + batch["y"]) / 2

    return batch
```

- Cada `batch` es un **DataFrame de pandas** con un subconjunto de filas.
- Tú solo te preocupas de transformar ese trozo:
    - Añadir columnas, calcular medias, etc.
- Ray se ocupará de:
    - pasar muchos batchs (lotes) por tu función en paralelo
    - distribuir el trabajo entre CPUs.
- Si el volumen de datos es muy grande y estás justo de RAM, puedes plantearte **no hacer** `.copy()` y asumir que modificas el batch "en sitio".
- A cambio, tienes que ir con más cuidado de no reutilizar ese mismo objeto en otros sitios.

3.7. Aplicar la transformación en paralelo

```
ds_enriched = ds.map_batches(
    enrich_batch,
    batch_format="pandas",
)
```

- `ds.map_batches(...)`:
    - Recorre el dataset bloque a bloque.
    - En cada bloque, llama a enrich_batch (en un worker de Ray).

- Devuelve un **nuevo Ray Dataset** con las nuevas columnas añadidas.

- No hay un solo DataFrame gigante: Ray mantiene los datos en **particiones**.

3.8. Agregados globales (ej. medias)

```
if "z" in ds_enriched.schema().names:
    mean_z = ds_enriched.mean("z")
    print(f"\nMedia global de z: {mean_z}")
```

- `ds_enriched.mean("z")`:
  - Calcula la media de la columna `z` **de forma distribuida**.
  - Ray:
    - calcula medias parciales en cada bloque,
    - luego las combina.
- No necesitas traer todo a pandas para esto.

Lo mismo con `xy_mean` u otras columnas numéricas.

3.9. Guardar el resultado en parquet

```
salida_dir = r"E:\datos\salida_enriquecida"
os.makedirs(salida_dir, exist_ok=True)


ds_enriched.write_parquet(salida_dir)
```

- `write_parquet` crea **múltiples ficheros parquet** en la carpeta:
  - Algo tipo: `part-00000.parquet`, `part-00001.parquet`, …
- De nuevo, no hay un "megafichero" único en memoria:
  - Se van escribiendo particiones desde los workers.

En la práctica es mucho más cómodo trabajar luego con esos parquet particionados (puedes leerlos con Ray, Pandas, Dask, Spark, etc.).

Nota:

- Trabajar con Ray leyendo desde USB está bien para pruebas.
- **Escribir 50 GB procesados a USB puede ser bastante lento**.
- Mejor salida en SSD interno (por ejemplo C:\datos\…) salvo que el USB sea muy rápido.

3.10. ¿Y si mi JSON NO es JSON Lines?

Si tu archivo archivo50G.json es así:

```
[
  {"x": 1, "y": 2},
  {"x": 3, "y": 4},
  ...
]
```

es decir, un **array gigante** de objetos JSON, entonces:

- Leer eso en streaming es complicado (tienes que parsear corchetes, comas, etc.).
- Casi siempre se recomienda **convertirlo una vez** a JSON Lines o Parquet antes de hacer cosas gordas.

Por ejemplo, con un script de conversión que lea poco a poco y vaya escribiendo línea a línea.

Si quieres, en el siguiente paso podemos:

- Ajustar el código a la **estructura real** de tu JSON (si me dices qué campos tiene).
- O hacer una "mini versión" con un JSON pequeño de ejemplo antes de soltarle los 50 GB, para que puedas probar que todo funciona sin esperar una eternidad.

4. Generar datos de prueba con Faker

Crear un faker.py para generar el archivo json en mi usb, y poder probar el script que hemos generado.

Antes de nada: **no llames al archivo faker.py** si vas a usar la librería faker, porque entonces Python intentará importar *tu* fichero en vez del paquete faker. Llámalo, por ejemplo, generar_json_falso.py.

4.1. Instalar Faker

En tu entorno (el mismo donde usas Ray), ejecuta:

```
pip install faker
```

4.2. Script generar_json_falso.py

Guarda esto en un fichero, por ejemplo en tu escritorio o donde quieras:

```
import os
import json
```

```
import time
from faker import Faker
import random

# ---------------- CONFIGURACIÓN ----------------

# Ruta de salida (ajústala a la ruta de tu USB)
OUTPUT_PATH = r"E:\datos\archivo50G.json"

# Tamaño objetivo aproximado (en GB)
TARGET_SIZE_GB = 1  # pon 50 cuando ya lo tengas probado
# -----------------------------------------------


def generar_registro(fake: Faker) -> dict:
    """Devuelve un diccionario con datos ficticios para una línea JSON."""
    return {
        "id": fake.uuid4(),
        "nombre": fake.name(),
        "email": fake.email(),
        "fecha_registro": fake.iso8601(),
        "pais": fake.country(),
        "x": random.uniform(-1000, 1000),
        "y": random.uniform(-1000, 1000),
        "comentario": fake.text(max_nb_chars=200),
    }


def main():
    fake = Faker()
    target_bytes = int(TARGET_SIZE_GB * (1024**3))  # GB -> bytes

    # Crear carpeta destino si no existe
    os.makedirs(os.path.dirname(OUTPUT_PATH), exist_ok=True)

    print(f"Generando archivo JSON lines en: {OUTPUT_PATH}")
    print(f"Tamaño objetivo aproximado: {TARGET_SIZE_GB} GB ({target_bytes} bytes)\n")

    start_time = time.time()
    num_registros = 0

    # Abrimos el fichero en modo texto y escribimos línea a línea
    with open(OUTPUT_PATH, "w", encoding="utf-8") as f:
        while True:
            registro = generar_registro(fake)
            linea = json.dumps(registro, ensure_ascii=False)
            f.write(linea + "\n")
            num_registros += 1

            # Cada X registros comprobamos tamaño y mostramos progreso
            if num_registros % 10_000 == 0:
                f.flush()
                os.fsync(f.fileno())

                size = os.path.getsize(OUTPUT_PATH)
                elapsed = time.time() - start_time
                mb = size / (1024**2)
                gb = size / (1024**3)

                print(
                    f"Registros: {num_registros:,} | "
                    f"Tamaño aprox: {mb:,.2f} MB ({gb:.2f} GB) | "
                    f"Tiempo: {elapsed:.1f} s",
                    end="\r",
                )

                if size >= target_bytes:
                    break

    elapsed = time.time() - start_time
    final_size = os.path.getsize(OUTPUT_PATH) / (1024**3)
    print("\n\n¡Terminado!")
```

```
    print(f"Registros generados: {num_registros:,}")
    print(f"Tamaño final aprox: {final_size:.2f} GB")
    print(f"Tiempo total: {elapsed:.1f} s")


if __name__ == "__main__":
    main()
```

Nota:

El `flush` y `fsync` cada 10 000 registros garantizan que el progreso y el tamaño sean bastante precisos, pero si ves que va muy lento, puedes subir ese número (100 000, 1 000 000) para reducir el número de escrituras forzadas a disco.

4.3. Cómo usarlo

1. Ajusta en el script:
    - OUTPUT_PATH → ruta real de tu USB, por ejemplo:

      `OUTPUT_PATH = r"F:\archivo50G.json"`
    - TARGET_SIZE_GB:
        - Empieza con 0.1 (100 MB) o 0.5 (500 MB) para probar.
        - Cuando veas que todo va bien, súbelo a 50.
2. En la carpeta donde guardaste el script:

   `python generar_json_falso.py`
3. Verás algo tipo:

   `Registros: 120,000 | Tamaño aprox: 95.23 MB (0.09 GB) | Tiempo: 12.3 s`

Hasta que llegue al tamaño objetivo.

4.4. Usarlo con el script de Ray

Una vez tengas tu archivo50G.json (o uno más pequeño de prueba):

- Usa la ruta en el script de Ray que te di antes:

  `ruta_archivo = r"E:\datos\archivo50G.json"`

Y ya estarás procesando datos “de mentira” pero con el mismo formato que usarías en producción.

Si quieres, en el siguiente paso podemos ajustar el enrich_batch para usar estos campos (nombre, pais, x, y, etc.) y hacer, por ejemplo, medias por país o cosas así.